VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# OPTIMALIZACE V PŘEKLADAČI C PRO VLIW ARCHITEKTURY

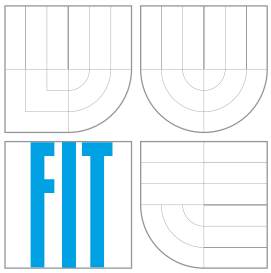
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RÓBERT BARUČÁK

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# OPTIMALIZACE V PŘEKLADAČI C PRO VLIW ARCHITEKTURY

OPTIMIZATIONS IN C COMPILER FOR VLIW ARCHITECTURES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RÓBERT BARUČÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. KAREL MASAŘÍK, Ph.D.

BRNO 2014

# Abstrakt

Prezentován je implementovaný algoritmus alias analýzy, který byl následně integrován s frameworkem LLVM. Diskutovány jsou vlastnosti a limity různých algoritmů. Dále jsou demonstrovány rozličné přístupy k práci s predikovanými instrukcemi a jejich integrace s LLVM. Jedním z výsledků diplomové práce je i návrh a implementace profilem řízené if-konverze.



# Abstract

Presented is implementation of algorithm for alias analysis, which was integrated into LLVM framework. Properties and limitations of various alias analysis algorithms are discussed. Demonstrated are different approaches to working with predicates and integration of these principles with LLVM. One of the outcomes of this master's thesis is design and implementation of algorithm for profile guided if-conversion.



# Klíčová slova

LLVM, překladač, VLIW, optimalizace, alias analýza, if-konverze, predikáty, Codasip

# Keywords

LLVM, compiler, VLIW, optimization, alias analysis, if-conversion, predicates, Codasip


# Citace

Róbert Baručák: Optimalizace v překladači C pro VLIW architektury, diplomová práce, Brno, FIT VUT v Brně, 2014

# Optimalizace v překladači C pro VLIW architektury

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doktora Masaříka

.....
Róbert Baručák
12. ledna 2014

## Poděkování

Ďakujem pánovi inžinierovi Husárovi a pánovi doktorovi Masaříkovi za konzultácie a odbornú pomoc. Ďalej chcem poďakovať celému týmu Codasip.

# Obsah


**1 Úvod** **3**

**2 Využité nástroje** **4**
2.1 Codasip . . . 4
2.1.1 Codal . . . 5
2.1.2 Súčasti frameworku . . . 5
2.2 Prekladový systém LLVM . . . 5
2.3 Vnútorná reprezentácia kódu . . . 6
2.3.1 Trojadresný kód . . . 6
2.3.2 SSA forma . . . 7
2.4 LLVM priechody . . . 7
2.5 Generovanie kódu pomocou LLVM backendu . . . 8

**3 VLIW architektúra** **9**
3.1 Charakteristika . . . 9

**4 Alias analýza** **11**
4.1 Charakteristika . . . 11
4.2 Vlastnosti alias analýz . . . 12
4.2.1 Model programu . . . 13
4.2.2 Známe algoritmy . . . 13
4.3 Zvolený algoritmus . . . 15
4.3.1 Charakteristika . . . 15
4.3.2 Vlastné vylepšenia zvoleného algoritmu . . . 16
4.4 Implementácia a zmeny oproti pôvodnému algoritmu . . . 17
4.4.1 LLVM priechod . . . 17
4.4.2 LLVM a alias analýza . . . 19
4.4.3 Vlastná implementácia . . . 21

**5 Predikované inštrukcie** **25**
5.1 Prístupy k predikácii inštrukcií . . . 25
5.1.1 Plná predikácia . . . 26
5.1.2 Čiastočná predikácia . . . 26
5.1.3 Prínos predikácie . . . 27
5.2 If-konverzia . . . 28
5.2.1 LLVM a if-konverzia . . . 28
5.2.2 Profilom riadené optimalizácie . . . 28
5.2.3 Návrh a implementácia . . . 30

**6 Dosiahnuté výsledky** **33**
6.1 Alias analýza . . . . . 33
6.1.1 Presnosť . . . . . 33
6.1.2 Urýchlenie výsledného kódu . . . . . 36
6.2 If-konverzia . . . . . 36

**7 Záver** **37**

# Kapitola 1

# Úvod

Význam vstavaných systémov sa neustále zväčšuje. Tlak na výkonnejšie, menšie a úspornejšie procesorové systémy dáva znova priestor na rozvoj nie úplne tradičným architektúram. Požiadavky dizajnérov na malý príkon, vysokú mieru inštrukčného paralelizmu a teda aj výkon ich privádzajú k architektúre VLIW.

VLIW prístup umožňuje velké zjednodušenie dizajnu oproti klasickým superskalárnym architektúram. Využívanie paralelizmu na inštrukčnej úrovni je v prípade superskalárnych architektúr vykúpené komplikovaním dizajnu, nakoľko sa rieši na hardwarovej úrovni. U VLIW je inštrukčný paralelizmus riešený prekladačom, čo umožňuje nižší príkon pri ekvivalentnom výkone.

Riešenie inštrukčného paralelizmu vytvára požiadavky na kvalitu použitého prekladového systému. Aj keď otázka miery zvýšenia náročnosti oproti prekladovým systémom pre superskalárne architektúry je otvorená [4].

Pri preklade pre VLIW architektúry sa vo väčšej miere uplatňujú optimalizačné algoritmy súvisiace s preskladávaním inštrukcií. Preto je výhodné aby mal prekladač presné informácie o tom, ako daný program pristupuje k pamäti. Použitie správneho algoritmu alias analýzy má veľký dopad na kvalitu výstupného kódu. V texte sú analyzované rozličné prístupy k alias analýze a následne je predstavený implementovaný algoritmus.

So zvyšujúcou sa mierou paralelizmu na inštrukčnej úrovni je viditeľné ďalšie úzke hrdlo pre optimalizovanie. Sú ním inštrukcie vetvenia programu. Bežný program, podľa [4], obsahuje vetvenie každých 6 až 7 operácií. Odpoveďou na tento výkonový problém je zavedenie predikovaných inštrukcií a if-konverzie. Uvedené opatrenia vedia zväčšiť vzdialenosť vetviacich inštrukcií až na každých 8 až 10 operácií. Vďaka predikovaným inštrukciám teda môže vzniknúť väčší priestor pre paralelizmus na inštrukčnej úrovni.

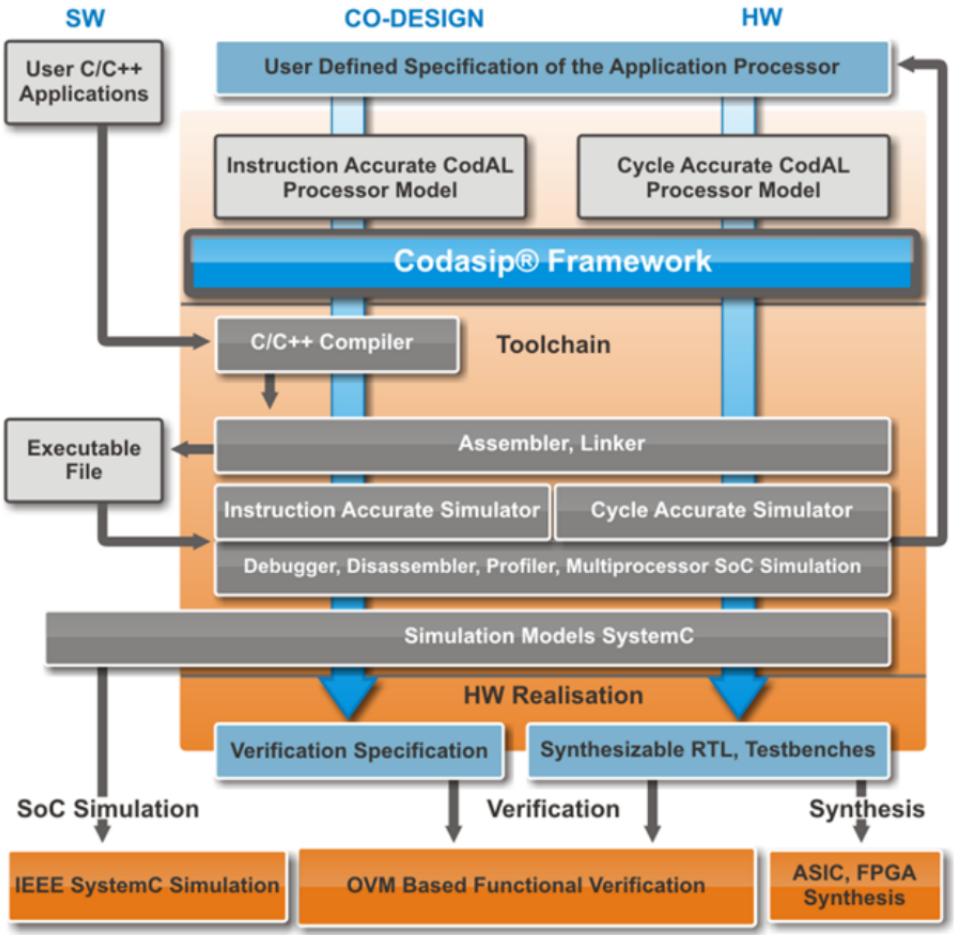
Cieľom diplomovej práce je analyzovanie, navrhnutie a implementácia vybraných optimalizačných techník. Následne oboznamuje čitateľa s efektom využitia týchto techník v modernom prekladovom systéme. Motiváciou je podstatné vylepšenie vlastností generovaného kódu v prostredí produktu Codasip.

# Kapitola 2

# Využité nástroje

## 2.1 Codasip

Codasip framework predstavuje integrované vývojové prostredie určené na vytváranie aplikačne špecifických procesorov. K dispozícii je široká sada nástrojov určených pre podporu vývoja [1].




Obrázek 2.1: Codasip framework [1]

### 2.1.1 Codal

Na popis aplikačne špecifických procesorov sa využíva jazyk Codal. Ide o hierarchický, štruktúrovaný jazyk určený na popis architektúry, ktorý sa využíva na reprezentovanie procesorovcýh jadier na vyššej úrovni abstrakcie. Výhodou jazyka Codal je vyššia úroveň, na ktorej sa vykonáva popis, čo umožňuje rýchle prototypovanie architektúr. Popis zvláda nielen RISC architektúry (napr. ARM, MIPS), ale aj modelovanie CISC architektúr (x86). Možné je aj modelovanie neštandartných prvkov VLIW architektúr, ako je napríklad bundling a inštrukčná kompresia.

### 2.1.2 Súčasti frameworku

Codasip framework poskytuje komplexnú sadu nástrojov určených na podporu dizajnovania procesorov.

- debugger, simulátor – ladenie optimalizovaného kódu je zložitý proces, nakoľko optimalizujúci prekladač C preskladáva inštrukcie (lepšie využitie pipeline a pod.). Preto väčšinou neexistuje vzťah medzi vygenerovaným kódom a C zdrojovým kódom. Na ladenie už optimalizovaného kódu sa používa disassembler. Pre ladenie neoptimalizovaného spustitelného kódu sa využíva n postavený na platforme GDB.

  Podstatnou časťou debuggeru je simulátor, ktorý umožňuje užívateľovi simulovať kód pre rozličné architektúry. V základe sa využíva interpretovaná simulácia.

- profiler – je určený na získavanie informácií o procese simulácie. Sleduje a zaznamenáva informácie počas simulácie. Napríklad najpoužívanejšie inštrukcie, nepoužité inštrukcie, graf volaní funkcií s počtom cyklov, využité zdroje atd. Vďaka týmto informáciám sa dá optimalizovať nielen architektúra, ale aj program spúšťaný na nej.

- prekladač jazyka C – framework ponúka prekladač C založený na frameworku LLVM, prekladač zvláda riešenie množstva inštrukčných hazardov automaticky. Je generovaný pre každú architektúru automaticky. Generovanie prebieha v dvoch krokoch, najprv je nutné z modelu procesoru vyextrahovať sémantiku inštrukcií, následne sa vygeneruje prekladač.

- Codasip studio – prostredie postavené na opensource projekte Eclipse, poskytuje rozhranie pre vývoj modelov v jazyku Codal a zvyšné časti frameworku.

Framework ponúka aj funkčnú verifikáciu, generovanie hardwaru, možnosti vytvárania komplexných SoC a pod.

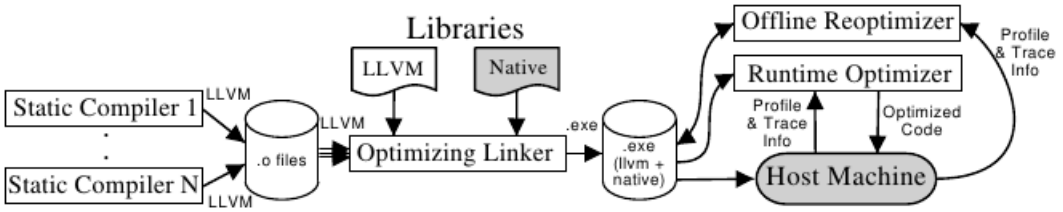
## 2.2 Prekladový systém LLVM

Framework Low Level Virtual Machine predstavuje moderný, modulárny prekladový systém. Platforma obsahuje kolekciu nástrojov a knižníc určených pre stavbu prekladačov, optimalizátorov a rozličných JIT generátorov.

LLVM začal ako výskumný projekt na University of Illinois. Distribuuje sa s upravenou BSD licenciou a našiel široké uplatnenie v komerčnej aj akademickej sfére. Cieľom bolo vytvoriť moderný prekladový systém schopný spracovávať statickú a dynamickú kompiláciu pre viaceré programovacie jazyky.

Spracovávanie vstupného kódu sa vykonáva pomocou frontendov, ktoré staticky preložia vstupný kód. Následne emitujú kód vo formáte virtuálnej inštrukčnej sady (LLVM IR). Táto vnútorná reprezentácia je nezávislá na cieľovej architektúre a v súčasnosti existuje široká škála frontendov podporujúcich mnohé jazyky (C,C++,Java, Haskell etc.).

Po vygenerovaní medzikódu nasleduje aplikácia optimalizačných priechodov. Všetky optimalizácie, transformácie a analýzy dostupné nad LLVM medzikódom sú dostupné pomocou programu *Opt*. O prevod IR kódu do strojového jazyka cieľovej architektúry sa starajú backendy. Informácie o architektúre LLVM som čerpal najmä z [3] a [10].




Obrázek 2.2: Prekladová schéma LLVM [3]

## 2.3 Vnútorná reprezentácia kódu

Virtuálna inštrukčná sada je navrhnutá tak, aby bola nízkoúrovňovou reprezentáciou s podporou vysokoúrovňových analýz a transformácií. Aby bolo možné dosiahnuť požadované správanie, táto sada poskytuje extenzívne jazykovo a architekturálne nezávislé typové informácie.

IR reprezentuje virtuálnu architektúru, ktorá zachytáva kľúčové operácie bežných procesorov. Zároveň sa však vyhýba špecifickým vlastnostiam, ako napríklad fyzickým registrom, zreťazeným inštrukciám alebo nízkoúrovňovým volacím konvenciám. LLVM poskytuje neobmedzenú sadu typovaných virtuálnych registrov.

Na premiestňovanie hodnôt z virtuálnych registrov do pamäte a vice versa slúžia výhradne operácie *load* a *store*. Alokovanie dát na zásobníku prebieha pomocou funkcie *alloca*. *Getelementptr* sa využíva na získavanie adries prvkov agregovaných dátových typov. Keďže LLVM definuje virtuálnu inštrukčnú sadu, táto neponúka žiadne I/O funkcie ani funkcie spojené s OS.

### 2.3.1 Trojadresný kód

Väčšina operácií je vo forme trojadresného kódu, všetky aritmetické a logické operácie majú jeden alebo dva operandy a jediný výsledok. Podstatnou výnimkou sú funkcie volacie a *phi* funkcia, ktorá súvisí s SSA formou.

IR disponuje sadou bežných aritmetických a logických operácií: add, fadd, sub, fsub, udiv, sdiv , fdiv , shl, or, xor atd.. Inštrukcie sú polymorfické, takže typ operandu zároveň definuje sémantiku inštrukcie a typ výsledku. Inštrukčná sada je silne typovaná. Podrobnejšie informácie o IR sú uvedené v referenčnom manuáli [11].

### 2.3.2 SSA forma

Program je v *static single assignment* (SSA) forme, ak každá premenná je definovaná presne jedenkrát a každé jej použitie nasleduje (je dominované) touto definícou. Daný spôsob reprezentácie je výhodný pre mnohé typy optimalizácie kódu. Podľa [15] prináša aj výrazné uľahčenie analýzy dátového toku. Preto LLVM využíva SSA ako primárnu formu reprezentácie kódu.

V praxi sa to prejavuje implicitným vytvorením virtuálneho registra pre každú inštrukciu, ktorá vracia hodnotu. Ak meno nie je stanovené v zdrojovom kóde, vygeneruje sa unikátne. Vďaka tomuto správaniu je vždy možné jednoducho priradiť definíciu ku každému použitiu hodnoty, vzniká takzvaný *use-def* reťazec.

Aby bolo možné zvládať riadiace štruktúry, ako napríklad cykly a podmienky, bolo nutné zaviesť $\phi$ funkciu. Jej výsledná hodnota je závislá od základného bloku, ktorý predal kontrolu. Na vstupe má dvojice základný blok a premenná. Musí byť vždy uvedená na začiatku základného bloku.

## 2.4 LLVM priechody

Priechody slúžia na vykonávanie transformácií a analýz v frameworku LLVM. Sú základnou jednotkou štruktúrovania funkcionality v LLVM.

K dispozícii sú mnohé predpripravené statické analýzy a transformácie. Implementované priechody sú k dispozícii v troch hlavných kategóriách.

*Analysis* priechody získavajú informácie o programe využiteľné v iných priechodoch. Môžu slúžiť pre vizualizáciu a odhaľovanie chýb. *Transforms* využívajú a prípadne invalidujú *analysis* priechody. Všetky *transforms* istým spôsobom modifikujú zdrojový kód. Posledným typom sú *utility* priechody. Vždy poskytujú funkčnosť, ktorá nespadá pod predchádzajúce kategórie. Typickým príkladom je priechod, ktorý extrahuje funkciu do bitového kódu. Nasleduje stručný popis niekoľkých kľúčových transformácií, z ktorých mnohé priamo závisia na kvalite informácií dodávaných alias analýzou.

- *Mem2reg* – transformácia sa používa na propagovanie odkazov do pamäte na prítupy do registrov.
- *Loop-simplify* – zabezpečuje kanonikalizáciu cyklov pomocou vkladania nových vstupných a výstupných základných blokov do cyklov. Unikátny vstupný blok cyklu garantuje, že existuje jediná vstupná hrana pre cyklus prístupná mimo cyklu samotného. Vloženie nového výstupného bloku umožňuje vznik jedinej výstupnej hrany pre cyklus. Táto transformácia značne zjednodušuje neskoršiu manipuláciu s telom cyklu.
- *Indvars* – vykonáva niekoľko zmien v cykloch programu. Prevedie všetku ukazatelovú aritmetiku na indexovanie poľa. Zabezpečuje, že indukčná premenná začína na nule a je inkrementovaná vždy len o jedna.
- *Loop-unroll* – transformácia, ktorá má na starosti rozbalovanie cyklov.
- *Loop Invariant Code Motion* – optimalizačná transformácia, ktorá má za úlohu redukovanie počtu inštrukcií v telách cyklov. Využíva informácie dostupné z alias analýzy tak aby mohla identifikovať kód, ktorý je invariantný v rámci cyklu.
- *Inline* – priechod, ktorý vkladá telá funkcií namiesto volaní.

- *Lowerswitch* – prevádza switch inštrukciu na vetvenie. Komplikuje kód, pre každú vetvu vytvára seprarátny základný kód.
- *Simplifycfg* – transformácia eliminujúca mŕtvy kód. Odstraňuje základné bloky bez predchodcu. Ak má blok len jednoho nasledovníka a nasledujúci blok má len jednoho predchodcu, vykoná zlúčenie.

Zoznam všetkých dostupných priechodov s ich popisom je dostupný na stránke [17].

## 2.5 Generovanie kódu pomocou LLVM backendu

Jednou z najväčších výhod systému LLVM je možnosť generovania kódu pre rozličné platformy. Postup generovania kódu je nezávislý na cieľovej architektúre. Informácie o generovaní kódu sú dostupné na [8]. Generovanie kódu prebieha v niekoľkých fázach.

- Selekcia inštrukcií – pri generovaní kódu sa z inštrukcií LLVM IR vytvára stromová reprezentácia nazývaná DAG (Directed acyclic graph). Uzly grafu zahŕňajú operačný kód, operandy a výsledok inštrukcie. V strome sú reflektované operačné a dátové závislosti.
- Legalizácia – DAG je legálny, ak všetky dátové typy a inštrukcie, ktoré sa v ňom vyskytujú, sú podporované aj cieľovou architektúrou. Konvertuje dátové typy na typy natívne podporované cieľovou architektúrou. Informácie o cieľovej architektúre sú odvodzované z popisu jej registrov. Legalizácia operácií zaručuje prítomnosť operácií natívnych pre cieľovú architektúru. V backende musia byť popísané vzory jednotlivých podporovaných operácií.
- Plánovanie inštrukcií – priradí legálnemu DAGu poradie jednotlivých inštrukcií. Následne sa DAG rozbíja, zoznam cieľových inštrukcií je stále v SSA forme. Poradie inštrukcií je volené podľa preferencií jednotlivých architektúr.
- Alokácia registrov – pozostáva z mapovania programu, ktorý má neobmedzené množstvo virtuálnych registrov na program, ktorý má obmedzené množstvo fyzických registrov. Využívajú sa rozličné techniky farbenia grafu. Odstraňuje všetky referencie na virtuálne registre. Ak počet fyzických registrov nie je postačujúci pre algoritmus alokácie, tak sú niektoré z virtuálnych mapované do pamäte. Tento postup je známy aj ako *spilling*. Alokácia registrov má viacero módov s rozličnými vlastnosťami.
- Vkladanie prológu a epilógu – po úspešnom nagenerovaní strojového kódu a namapovaní registrov je možné vypočítať veľkosť miesta, ktoré zaberú jednotlivé funkcie na zásobníku. Do funkcií sa vloží prológ a epilóg. Abstraktné ukazatele do zásobníku sú nahradené fyzickými adresami.
- Emitovanie kódu – posledným krokom je emitovanie kódu vo forme assembleru danej architektúry.

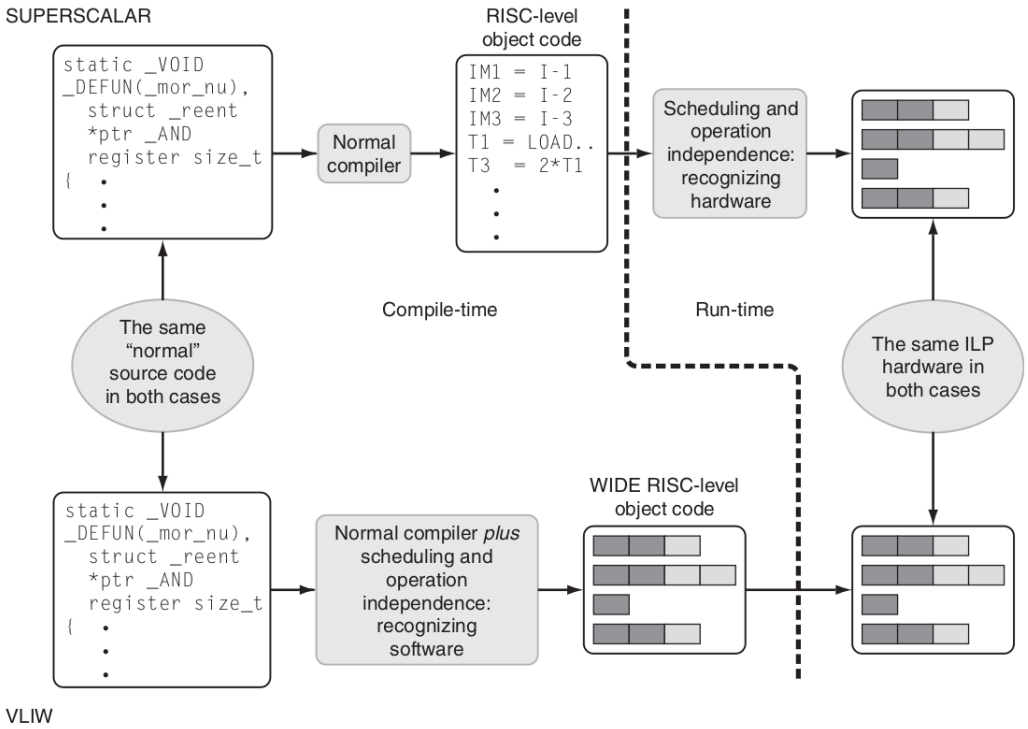
# Kapitola 3

# VLIW architektúra

## 3.1 Charakteristika

Základnou myšlienkou architektúry je sprístupnenie inštrukčného paralelizmu. Od návrhu hardwaru, cez inštrukčnú sadu, až po kompilátor určený pre danú platformu. Z historického hladiska sa jedná o vysokoúrovňovú general-purpose architektúru.

Podľa [4] hlavný rozdiel oproti superskalárnym architektúram predstavuje myšlienka, že procesor nevytvára paralelizmus na inštrukčnej úrovni (ILP) bez priamej špecifikácie v strojovom kóde.




Obrázek 3.1: VLIW a superskalárne architektúry. Hardware vykonávajúci ILP je približne ekvivalentný. Rozdiel je v plánovaní inštrukcií, ktoré prebieha u VLIW-ov na strane prekladača, zatial čo u superskalárnych architektúr priamo v hardware [4].

Principiálne sa dajú pravidlá pre tvorenie VLIW dizajnu zhrnúť do niekoľkých bodov:

- Neumožňovať hardwaru vykonávať veci, ktoré nie sú kontrolované strojovým kódom (z hladiska ILP) – teda zmenšovať komplexnosť hardwaru.
- Snaha o vyhýbanie sa hardwaru, ktorý vykonáva inú činnosť ako zamýšlaný výpočet v kritickej ceste inštrukcie.

Prvotný vývoj a aplikáciu VLIW architektúry datujeme do 80. rokov 20. storočia. Prístup bol využitý pri návrhu superpočítačov firmami Multiflow a Cydrome. Obidva dizajny predstavovali komerčný neúspech. Väčšiemu uplatneniu zabránili komplikovanosť hardwaru oproti dobovým RISC architektúram a kvalita dostupných prekladových systémov. Problémom vždy bola aj modifikovateľnosť dizajnov a spätná kompatibilita nakoľko implementačné detaily sú prístupné inštrukčnej sade. Ak by sa do hypotetického dizajnu pridala ďalšia výpočtová jednotka, tak sa zvýši množstvo operácií, ktoré je možné vykonávať paralelne. Zároveň sa však zväčší dĺžka inštrukčného slova, takže starý binárny kód už nie je spustiteľný na upravenej architektúre.

Medzi nevýhody treba zahrnúť aj zväčšenie kódu (nevyužité bity v inštrukčnom slove). Väčšie množstvo registrov je nutné kvôli viacerým výpočtovým jednotkám. Nezanedbateľný je aj nárast komunikácie medzi inštrukčnou cache a fetch jednotkou.

Najväčšou výhodou je možnosť uvedenia viacerých výpočtových jednotiek, bez nutnosti zavádzať komplikovaný hardware určený na detekciu paralelizmu. Zároveň majú prekladače väčší priestor pre uplatnenie optimalizácií, ktoré prebiehajú globálne nad celým kódom.

# Kapitola 4

# Alias analýza

## 4.1 Charakteristika

Analyzovanie programov napísaných v jazykoch obsahujúcich ukazatele vyžaduje informácie o správaní sa ukazateľov. Bez týchto znalostí sa musíme riadiť konzervatívnymi odhadmi ohľadom prístupov k pamäti, čo nepriaznivo ovplyvňuje efektivitu a presnosť analýz využívajúcich tieto informácie. Spravidla sa jedná o analýzy optimalizujúcich prekladačov.

**Príklad 1** Príklad aliasovania

```
p = new Object;
q = p;
```

*q a *p sa prekrývajú.

Typicky sa pokúša o získanie odpovede na otázku „Ukazujú dva rozdielne ukazatele na jedno miesto v pamäti?“. Jedná sa o algoritmy vystavávajúce množiny lokácií v pamäti, na ktoré môže ukazateľ v čase behu programu ukazovať. Praktický význam predstavuje schopnosť identifikovať miesta, ktoré sú počas behu programu referencované a modifikované. Algoritmy riešiace uvedený problém sú v literatúre označované súhrnne pod pojmom *Alias analýza*. Poskytuje prekladaču informácie o miere nezávislosti jednotlivých príkazov programu. Vďaka týmto informáciám je základom pre rozličné optimalizácie, automatické paralelizačné nástroje a plánovače využívajúce paralelizmus na inštrukčnej úrovni.

Podľa [18], pojem alias spadá do kategórie vlastností ako sú *zdieľanie* a *disjunktnosť*. Sú vzájomne súvisice, ale neoznačujú tú istú vlastnosť. Všetky tri sú definované nad rozličnými doménami a sú využívané pri riešení rozličných úloh prekladača.

Vlastnosť *alias* sa týka dvojíc ukazateľov, pričom je relevatná pri vyhodnocovaní prístupov do pamäte. *Zdieľanie* sa týka susediacich jednotiek na hromade, využíva sa pri riadení prístupov do pamäte v čase prekladu. Pod *disjunkciou* rozumieme vzťah medzi pármi dátových štruktúr a určuje, či môžu operácie nad nimi paralelizované.

Aj keď uvedené vlastnosti nie sú totožné, existuje medzi nimi úzke prepojenie, ktoré sa v aplikáciách využíva.

- *Disjunkcia-alias* – Dve dátové štruktúry $D_1$ a $D_2$ sú disjunktné v každom stave ak neexistujú ukazatele $e_1$, ukazujúci na $D_1$ a $e_2$, ukazujúci na $D_2$ také, že by sa aliasovali v ľubovolnom stave.

- *Disjunkcia-zdieľanie* – Ak dve dátové štruktúry $D_1$ a $D_2$ nie sú disjunktné v nejakom

stave, tak aspon jeden z elementov $D_1$ a $D_2$ je zdieľaný práve v tomto stave.

- *Alias-zdieľanie* – Ak dva rozličné ukazatele $e_1$ a $e_2$ odkazujú na rovnaké miesto v pamäti, tak toto miesto musí byť zdielané v danom stave programu.

V moderných prekladových systémoch, akým je napríklad LLVM, je alias analýza základom pre zavednie konceptu *závislosti*. Notácia *definície* a *používateľa* je nutnou podmienkou pre zavednie *závislosti*. Takáto analýza môže byť vykonaná na úrovni zdrojového kódu, kde predmetom záujmu sú najmä premenné v programe, ale aj na úrovni strojového kódu, kde máme k dispozícii informácie o registroch a pod.. Pre analýzy pracujúce nad úrovňou zdrojového kódu sú dané notácie generalizáciou skalárnych premenných a prvkov štruktúr. Definície a použitia v prípade skalárnych premenných jednoznačne vyjadrujú závislosť. Ak však sú generalizáciou prístupov k prvkom štruktúr, ako napríklad pole, tak vytvárajú len potencionálne závislosti. Pre kvalitu alias analýzy je kvôli tomu podstatné, akým spôsobom pristupuje k spracovaniu dátových štruktúr. Podrobnejšie je prístup k dátovým štruktúram popísaný v časti venovanej implementovaniu zvoleného algoritmu.

Výskumy a rozličné algoritmy riešiace problematiku alias analýzy sú uverejňované posledných 30 rokov, preto sa naskýta otázka položená v článku [7]: „Nevyriešili sme už tento problém?“. Aj napriek veľkému množstvu práce vykonanom v oblasti, ostáva množstvo otvorených otázok a prebieha neustále úsilie v práci na efektívnejsších algoritmoch ponúkajúcich veľkú presnoť.

## 4.2 Vlastnosti alias analýz

Alias analýza predstavuje vo všeobecnosti nerozhodnuteľný problém [16]. Bolo uverejnených veľké množstvo algoritmov, ktoré balancujú presnosť analýzy a efektivitu [16]. Zložitosti algortimov riešiacich alias analýzu sú v širokom rozsahu od skoro lineárnych [19] až po exponenciálne. Častokrát však *worst-case* časová zložitosť neindikuje správanie sa algoritmu v typických prípadoch, príkladom sú [14] a [13].

Existuje niekoľko základných kritérií, podľa ktorých sa dajú kategorizovať súčasné prístupy k alias analýze. Všetky majú významný dopad na presnosť a zložitosť daného algoritmu. Zároveň ovplyvňujú spôsob akým daná analýza modeluje spracovávaný program.

- Flow-sensitivity: Berie analýza do úvahy tok riadenia? Ak sa analýza nezaoberá tokom riadenia (je *flow-insensitive*) v procedúrach tak vytvára konzervatívne odhady. V tom prípade je schopná vytvoriť riešenie pre celý program, alebo pre každú procedúru. Naopak analýza, ktorá berie riadiaci tok do úvahy (je *flow-sensitive*) vytvára riešenie pre každý bod programu. Vo všeobecnosti sú analýzy zaoberajúce sa riadiacim tokom presnejšie ale zložitejšie. *Flow-insensitive* sa analýzy delia na *equality-based*, ktoré berú priradenia ako obojsmerné a typicky využívajú *union-find* dátové štruktúry. Druhým prípadom sú *subset-based*, ktoré berú priradenie ako jednosmerný tok hodnôt.

- Context-sensitivity: Analýza je považovaná za context-sensitive ak berie do úvahy volací kontext. Zároveň musí riešiť tok hodnôt od volania až po návrat z funkcie.

- Heap modeling: Delí analýzy na tie, ktoré nemodelujú haldu, pre každé miesto alokovania vytvoria jeden reprezentujúci objekt a tie, ktoré riešia zložitejšie *shape-analýzy*.

- Aggregate modeling: Delí analýzy podľa prístupu k modelovaniu agregovaných dátových štruktúr.

- Whole program: Pracuje analýza s celým programom alebo iba s časťou?

### 4.2.1 Model programu

#### *Points-to* relácia

Je relácia $pt : A \rightarrow 2^A$ kde A predstavuje množinu všetkých možných cieľov ukazateľov. Pre každý ukazateľ v programe poskytuje množinu cieľov na ktoré môžu ukazovať.

#### Obmedzenia

Alias analýzy pracujú s nasledujúcimi operáciami ovplyvňujúcimi ukazatele, ktoré sú označované ako obmedzenia. Všetky komplikovanejšie operácie s ukazateľmi sa dajú zjednodušiť na uvedené [7].

**Príklad 2** Typy obmedzení a ich vzťah k pt relácii. Tento konkrétny príklad ukazuje riešenie pt relácie pre Andersenovu analýzu.

| | |
|---|---|
| address: x = &y; | $\{y\} \supseteq pt(x)$ |
| copy: x = y; | $pt(y) \supseteq pt(x)$ |
| load: x = *y; | $\forall a \in pt(y).pt(a) \supseteq pt(x)$ |
| store: *x = y; | $\forall a \in pt(x).pt(y) \supseteq pt(a)$ |

### 4.2.2 Známe algoritmy

#### Andersenova analýza

Flow-insensitive, context-insensitive iteratívny algoritmus so zložitosťou $\mathcal{O}(N^3)$ [2]. Cieľom algoritmu je vytvorenie pt množiny pre každý ukazateľ v programe.

Postupne aplikujeme pravidlá uvedené v príklade 3, až kým sa nedá zmeniť žiadna pt množina.

**Príklad 3** Príklad aplikácie andersenovej analýzy.

| operácia | obmedzenia | riešenie |
|---|---|---|
| a = &b; | $pt(a) \supseteq \{b\}$ | pt(a) = {b, d} |
| c = a; | $pt(c) \supseteq pt(a)$ | pt(c) = {b, d} |
| a = &d; | $pt(a) \supseteq \{d\}$ | pt(b) = pt(d) = {} |
| e = a; | $pt(e) \supseteq pt(a)$ | pt(e) = {b, d} |

Príklad ilustruje aj flow-insensitivitu andersenovho algoritmu. Ak by bol algoritmus flow-sensitive riešenie by sa líšilo v pt(c) = {b} a pt(e) = {d}.

### Steensgardova analýza

Predstavuje často využívaný algoritmus ktorý je flow-insensitive a context-insensitive [19]. Je menej presnejší ako andersenov algoritmus, ale má významne redukovanú zložitosť $\mathcal{O}(N)$. Eliminuje nutnosť iterovania. Ak x ukazuje na y a z tak sú y a z ylúčené do triedy ekvivalencie. Zmena oproti andersenovmu algoritmu je vo vzťahu obmedzení a pt relácie.

**Príklad 4** Riešenie pt relácie pre Steensgardovu analýzu

| | |
|---|---|
| address: x = &y; | $\{y\} \supseteq pt(x)$ |
| copy: x = y; | $pt(y) = pt(x)$ |
| load: x = *y; | $\forall a \in pt(y).pt(a) = pt(x)$ |
| store: *x = y; | $\forall a \in pt(x).pt(y) = pt(a)$ |

Implementácia môže pracovať ako jeden priechodp programom. Využívaný je *union-find* pre udržiavanie a dopĺňanie pt množín.

**Príklad 5** Príklad ukazujúci rozdieľ v presnosti Steensgardovho a Andersenovho algoritmu

```
int **a, *b, c, *d, e;
1: a = &b;
2: b = &c;
3: d = &e;
4: a = &d;
```

Riešenia:

| Steensgardov algoritmus | Andersenov algoritmus |
|---|---|
| pt(a)= {b,c,d,e} | pt(a)= {b,c,d,e} |
| pt(b)= {c,e} | pt(b)= {c} |
| pt(c)= {} | pt(c)= {} |
| pt(d)= {c,e} | pt(d)= {e} |
| pt(e)= {} | pt(e)= {} |

Zjednotenie v Steensgardovom algoritme nastalo kvôli operácii č.4.

### DSA

*Data Structure Analysis* je context-sensitive, interprocedurálna analýza, ktorá modeluje hromadu pomocou techniky nazývanej *heap-cloning* [13]. Zároveň je schopná modelovať

agregované dátové štruktúry.

Heap-cloning v kontexte DSA znamená, že algoritmus identifikuje objekty vytvorené na hromade nie iba podľa bodu alokácie ale podľa acyklického grafu volaní daného objektu. Postup umožňuje pracovať s instanciami logických dátových štruktúr.

DSA algoritmus má niekoľko zaujímavých vlastností.

- Využíva rozšírenie Tarjanovho algoritmu SCC, ktoré umožňuje vytvorenie grafu volaní počas analýzy, inkrementálne bez iterácie nad programom.
- Vytvorená technika udržiavania pt množín umožňuje algoritmu pracovať aj nad nekompletnými programami (obsahujúce volania externých funkcií), zároveň umožňuje inkrementálne vytváranie grafu volaní.

Narozdiel od predchádzajúcich prístupov k modelovaniu programu, DSA algoritmus zavádza data structure graf. DS graf je vystavávaný pre každú funkciu v programe a zahŕňa pamäťové objekty ku ktorým môže funkcia pristúpiť počas behu. Všetky objekty, na ktoré môže byť odkazované jedným ukazateľom (alebo polom) sú reprezentované ako jeden uzol grafu. DS uzol reprezentuje potencionálne nekonečnú množinu objektov.

DS graf je definovaný ako štvorica $DSG(F) = (N, E, E_V, N_{call})$ kde

- $N$ je množina uzlov reprezentujúcich pamäťové objekty.
- $E$ predstavuje množinu hrán v grafe. E je reláciou $(n_s, f_s) \rightarrow (n_d, f_d)$ kde $n_s, n_d \in N, f_s \in pole(T(n_s))$ a $f_d \in pole(T(n_d))$ pričom $T(n)$ označuje odvodený typ. Dvojica (uzol,pole) sa označuje ako bunka.
- $E_V$ je parciálna funkcia $vars(f) \rightarrow (n, f)$ kde $vars(f)$ je množina virtuálnych registrov vo funkcii $f$. To zahŕňa aj globálne premenné, ktoré sa pokladajú za virtuálne registre ukazateľového typu, ukazujúce na nepomenovaný globálny pamäťový objekt. V praxi je $E_V(v)$ hrana z registru $v$ do cieľového pola $(n, f)$, ak $v$ má kompatibilný typ.
- $N_{call} \subset N$ je množina volacích uzlov v grafe, ktoré reprezentujú volania vo funkcii. Každý volací uzol $c \in N_{call}$ je $k+2$tica $(r, f, a_1...a_k)$ kde každý prvok je dvojica $(n, f)$. $r$ predstavuje hodnotu navrátenú volanou funkciou, $f$ predstavuje množinu možných volaných funkcií a $a_1...a_k$ sú hodnoty predávané parametrom.

Samotné vystavanie grafu pre program prebieha v troch krokoch. V prvom kroku sa intraprocedurálne vystavia samostatný graf pre každú funkciu. Iba v prvom kroku algoritmus pracuje priamo s kódom analyzovaného programu. Keď je graf pre každú funkciu vystavaný začne sa *bottom-up* fáza, ktorá zahrnie informácie z grafov volaných funkcií do grafov volajúcich funkcií. Posledná fáza *top-down*, dodá informácie z volajúcich funkcií volaným funkciám. Graf je korektný aj keď má funkcia zahrnuté informácie iba z podmnožiny funkcií, ktoré volá. Preto sa algoritmus jednoducho vysporiadava s nekompletnými programami.

Implementácia tejto analýzy bola dostupná v frameworku LLVM, ale bola už dávno odstránená. Obsahovala mnoho chýb a málo priechodov ju zvládalo využívať.

## 4.3 Zvolený algoritmus

### 4.3.1 Charakteristika

Jedná sa o variantu andersenovho algoritmu s niekoľkými zásadnými vylepšeniami, majúcimi vplyv na presnosť a zložitosť. Podľa [14] algoritmus predstavuje kompromis medzi náročnosťou flow-sensitive analýzy a efektivitou flow-insensitive analýzy. Zložitosť je $\mathcal{O}(EV^2)$

kde $V$ je množstvo ukazatelov v programe a $E$ je počet hrán v interprocedurálnom grafe toku riadenia.

#### Strong-updates

Ak v operácii priradenia vieme presne identifikovať premennú, do ktorej sa zapisuje, tak môžeme vytvoriť novú pt množinu pre danú premennú v bode vykonania priradenia. Tento postup u flow-sensitive analýz sa nazýva strong-updates.

Väčšina kódu prakticky používaných algoritmov je obsiahnutá v cykloch a bez strong-update vlastnosti majú flow-sensitive a flow-insensitive algoritmy približne rovnakú presnosť.

#### Flow-sensitívne obmedzenia

Notácia obmedzenia známa z andersenovho algoritmu je rozšíraná o identifikáciu miesta v programe, ktorému zodpovedádané obmedzenie.

#### Hybrid Cycle Detection a Hash-based Value Numbering

Algoritmus HVN má za cieľ zjednodušiť obmedzenia identifikované v programe. Pracuje v dvoch krokoch.

- Identifikuje top-level ukazatele, ktorých pt množiny sú ekvivalentné.
- Zlúči uzly grafu reprezentujúce tieto top-level ukazatele a pozmení všetky podmnožiny obmedzení tak, aby referovali na nový uzol.

HCD pracuje priamo počas fáze riešenia obmedzení. Predpokladajme, že existuje cesta $*p \rightarrow q \rightarrow ... \rightarrow r$ a existuje operácia $*p = r$. HCD vytvorí dátovú štruktúru reprezentujúce všetky takéto cesty v programe. Ak je počas riešenia obmedzení do pt(p) pridaná adresa a, tak vznikne cyklus $pt(a) \subseteq pt(q) \subseteq ... \subseteq pt(r) \subseteq pt(a)$. Ak vznikne takýto cyklus, tak môžeme zlúčiť všetky top-level ukazatele na ceste od $q$ po $r$.

### 4.3.2 Vlastné vylepšenia zvoleného algoritmu

Oproti klasickému Andersenovmu algoritmu bola navrhnutá sada úprav určených na zväčšenie presnosti analýzy. Úpravy sa dajú rozdeliť do dvoch častí, ale obidve vychádzajú z aplikovania algoritmu nad reprezentáciou kódu v LLVM IR a teda umožňuje špecifické zlepšenie.

#### Modelovanie dátových štruktúr

Slabinou Andersenovho algoritmu je, že nemodeluje žiadnym spôsobom dátové štruktúry. Zaoberá sa iba vlastnosťou *alias* a modeluje správanie jednotlivých ukazateľov ale ignoruje prístupy ku konkrétnej časti dátovej štruktúry. Takéto riešenie však značne redukuje presnosť algoritmu v praktickom nasadení.

Notácia ukazateľa v algoritme je doplnená o vybrané vlastnosti. Algoritmus teda vie, či ukazateľ ukazuje na jednoduchý alebo zložený dátový typ. Zároveň skúša vystavať informáciu o časti zloženého dátového typu, na ktorú ukazuje. Prienik pt množín dvoch ukazateľov ešte automaticky neznamená, že pre nich platí alias. Ak má algoritmus dosť informácií, tak vie určiť, že ukazatele nemôžu v danom bode programu ukazovať na identickú časť dátovej štruktúry.

#### Rozšírenie obmedzení o pravidlá špecificky konštruované pre LLVM IR

V základnej verzii algoritmu [14] sú nedostatočne prepracované vzťahy medzi obmedzeniami a inštrukciami LLVM IR. Týka sa to najmä inštrukcií, ktoré ovplyvňujú konštrukciu pt množín (*IntToPtr*, *Vaarg*, *GetElementPtr*). Detailnejšie sú uvedené rozšírenia a ich implementácia popísané v nasledujúcej časti textu.

## 4.4 Implementácia a zmeny oproti pôvodnému algoritmu

### 4.4.1 LLVM priechod

Ako každá analýza zaradená v LLVM je aj zvolený algoritmus implementovaný ako priechod. Jednou z výhod priechodového systému v LLVM je práve plánovanie a riadenie spúšťania jednotlivých priechodov, tak aby boli spúšťané čo najefektívnejšie a boli splnené všetky závislosti. Závislosti sú indikované aj triedou priechodov, ktorú dedí.

Nadtriedy priechodov sú konštruované práve podľa jednotiek programov, nad ktorými má priechod operovať. Je odporúčané zvoliť čo najšpecifickejšiu nadtriedu, aby mal systém dostatok informácií na optimálne zoradenie priechodov a teda urýchlenie behu samotného prekladača.

Ďalej medzi základné vlastnosti priechodov patrí, že môžu byť v istom bode automaticky spúšťané manažérom priechodov. V systéme sa dajú explicitne definovať závislosti od iných priechodov, vzniká tranzitívna relácia závislosti priechodov. Napríklad isté transformácie vyžadujú vytvorený graf volaní v programe, ktorý vystaváva na to určený priechod. Invalidácia je podstatnou explicitne definovanou vlastnosťou priechodov, každá transformácia musí určiť, aký typ analýz jej aplikácia invaliduje. Po aplikovaní transformácie LICM už nie je zabezpečená validnosť informácií dodávaných alias analýzou, preto je treba ju spustiť znova.

#### ImmutablePass

Trieda určená pre implementáciu priechodov, ktoré nepotrebujú byť spúšťané, nemenia svoj stav a nedajú sa invalidovať. Zároveň nemôžu nikdy invalidovať iný priechod. Nejedná sa teda o normálne analýzy ani transformácie. Aj keď je tento typ priechodov zriedkavo využívaný, je podstatný pri poskytovaní informácií o aktuálnom cieľovom stroji a rozličných iných statických informácií, ktoré môžu ovplyvniť transformácie.

#### ModulePass

Použitím *ModulePass* indikujeme, že priechod berie celý program ako jednotku spracovania. Spravidla pristupuje k funkciám v nepredvídateľnom poradí, pridáva alebo odoberá funkcie z programu. Keďže o jeho správaní nie je pre plánovač známe nič, tak nie je možné optimalizovať jeho beh.

Môže využívať priechody pracujúce nad funkciami pomocou *getAnalysis* rozhrania. Využitý priechod však nemôže byť ani tranzitívne závislý na žiadnom *ImmutablePass* a *ModulePass* priechode.

#### CallGraphSCCPass

Typ priechodu využívaný ak je nutné prejsť graf volaní programu v poradí zdola-hore. Sú dostupné mechanizmy pre stavbu a prechádzanie grafu volaní a manažér priechodov

má zároveň dosť informácií na to aby sa dalo vykonanie tohto priechodu optimalizovať. Implementácie musia spĺňať niekoľko podmienok.

- Priechod nemôže modifikovať ani pristupovať k funkciám iným ako je aktuálna silne spojená komponenta grafu, jej volaným funkciám a tým, ktoré ju priamo volajú.
- Musí zachovávať aktuálny objekt grafu volaní, teda musí v ňom reflektovať všetky zmeny vykonané v programe.
- Nemôže odstraňovať silne spojené komponenty grafu, avšak môže ich modifikovať.
- Môže pridávať a odstraňovať globálne premenné z modulu.
- Môže si udržiavať stav medzi zavolaniami manažérom priechodov.

#### FunctionPass

Na rozdiel od predchádzajúcich typov majú predvídateľné lokálne správanie. Je spúšťaný nad každou funkciou programu nezávisle od ostatných funkcií. Nie je zabezpečené poradie vykonávania týchto priechodov a nikdy nemodifikujú externé funkcie.

- Nemôžu modifikovať a pristupovať k inej ako aktuálne spracovávanej funkcii.
- Nemôžu pridávať a odstraňovať funkcie z aktuálneho modulu.
- Nemôžu pridávať a odstraňovať globálne premenné z aktuálneho modulu.
- Nemôžu si uchovávať stav medzi jednotlivými volaniami manažérom priechodov.

#### LoopPass

Priechod je vykonávaný nad každým cyklom vo funkcii nezávisle na ostatných cykloch. Cykly sú spracovávané v poradí zanorenia a to tak, že najvnútornejší je spracovaný ako prvý.

#### RegionPass

Je veľmi podobný *LoopPass*. Namiesto nad cyklami pracuje nad regiónmi kódu s jedným vstupom a jedným výstupným bodom. Poradie spracovávania zanorených regiónov je rovnaké ako pri *LoopPass*.

#### BasicBlockPass

Obmedzenia sú rovnaké ako pri *FunctionPass*, ale pracujú nad každým základným blokom, zároveň nemôžu vykonávať nasledujúce činnosti.

- Modifikovať a pristupovať k inému základnému bloku ako aktuálnemu.
- Udržiavať si svoj stav medzi volaniami manažérom priechodov.
- Modifikovať graf toku kontroly.

### 4.4.2 LLVM a alias analýza

*AliasAnalysis* trieda je primárne rozhranie, ktoré používajú implementácie alias analýzy a priechody využívajúce informácie poskytované alias analýzou. Rozhranie je navrhnuté tak aby bolo jednoducho možné zapojiť nové implementácie alias analýzy do frameworku LLVM. Informácie o LLVM rozhraní pre alias analýzy som čerpal z [9].

#### Reprezentácia ukazateľov

Rozhranie *AliasAnalysis* poskytuje niekoľko metód, ktoré sú určené na zadávanie dotazov ohladom aliasovania objektov v pamäti. Zároveň umožňuje zistiť či môže volanie funkcie modifikovať alebo čítať z objektu v pamäti.

Vo všetkých dotazoch sú pamäťové objekty reprezentované ako dvojica pozostávajúca z ich začiatočnej adresy (symbolická LLVM *Value**) a ich statickej veľkosti.

Reprezentovanie adresy a veľkosti zároveň je kritické pre korektnú alias analýzu.

**Príklad 6** Príklad ilustrujúci nutnosť uvedenia veľkosti objektu.

```
int i;
char C[2];
char A[10];
/* ... */
for (i = 0; i != 10; ++i) {
  C[0] = A[i];          /* One byte store */
  C[1] = A[9-i];        /* One byte store */
}
```

V prvom prípade alias analýza vyhodnotí, že ukladania do *C[0]* a *C[1]* sa neprekrývajú. Na základe tejto informácie môže analýza *Loop Invariant Code Motion* dať tieto dve ukladania mimo cyklus.

```
int i;
char C[2];
char A[10];
/* ... */
for (i = 0; i != 10; ++i) {
  ((short*)C)[0] = A[i];  /* Two byte store! */
  C[1] = A[9-i];          /* One byte store */
}
```

V druhom prípade sa však už ukladania prekrývajú, bez informácii o veľkosti objektu by však bolo nutné v obidvoch prípadoch predpokladať, že prekryv je možný.

#### Reprezentácia a reťazenie odpovedí

Primárnou metódou určenou na získavanie informácií z rozhrania je metóda *alias*. Vyžaduje aby dva vstupné ukazatele, ktoré sa vyhodnocujú, boli v rovnakej funkcii alebo aby aspon jeden z nich bol konštantný. Možné odpovede sú:

- *NoAlias* odpoveď je poskytnutá ak medzi poskytnutými ukazateľmi nie je žiadna závisloť. Klasickým prípadom je ak ukazujú oba na iný región pamäte. Avšak nezávislé

sú, aj keď sa oba využívajú iba na čítanie alebo medzi ich použitím dojde k uvoľneniu a realokácii pamäte.

- *MayAlias* odpoveď znamená možnosť prekrývanie ukazateľov. Typicky využitá aj pri nedostatku informácií.
- *PartialAlias* odpoveď je využitá ak nastáva prekryv, ale ukazatele nezačínajú na rovnakej adrese.
- *MustAlias* odpoveď je navrátená ak vieme garantovať, že dva ukazatele sú ekvivalentné.

Všetky implementácie alias analýzy využívajúce LLVM rozhranie sú zreťazené svojimi odpoveďami. Výnimku tvorí len základná *no-aa* implementácia. V praxi to znamená, že ak jedna implementácia nevie odpovedať, tak skúsi odpovedať ďalšia z použitých implementácií. Ak žiadna z dostupných implementácií nevie odpovedať využie sa najkonzervatívnejšia správna odpoveď teda *MayAlias* alebo *Mod/Ref*.

**Príklad 7** Príklad využitia zreťazenia

```
AliasAnalysis::AliasResult alias(const Value *V1, unsigned V1Size,
                                 const Value *V2, unsigned V2Size) {
  if (...)
    return NoAlias;
  ...

  // Couldn't determine a must or no-alias result.
  return AliasAnalysis::alias(V1, V1Size, V2, V2Size);
}
```

### Implementácia novej alias analýzy v prostredí LLVM

Hlavnou podmienkou pri implementovaní novej alias analýzy využívajúcej LLVM infraštruktúru je zvolenie správneho typu LLVM priechodu. Všetky alias analýzy, ktoré sú v súčastnosti k dispozícii v rámci projektu LLVM využívajú *ImmutablePass*. Dôvodom je, že žiadna z analýz neimplementuje algoritmus, ktorý by potreboval spracovávanie celého programu, vystavávanie množín a podobne. Vďaka tomu nie je nutné využívať priechody, ktoré by museli byť invalidované *PassManagerom*.

Ďalšie metódy rozhrania, ktoré by mala každá alias analýza implementovať sú:

- *pointsToConstantMemory* je metóda, ktorá zisťuje, či ukazateľ ukazuje na funkciu alebo globálnu konštantnú premennú.
- *doesNotAccessMemory* vyhodnocuje, či zadaná funkcia môže čítať alebo zapisovať do pamäte. Podmienka je splnená pre funkcie bez vedľajších efektov, môžu pristupovať ku konštantnej pamäti.
- *onlyReadsMemory* zisťuje, či daná funkcia číta iba z nevolatilnej pamäti.

### 4.4.3 Vlastná implementácia

Aj napriek prepracovanosti LLVM AliasAnalysis rozhrania existujú nezanedbateľné limity, ktoré obmedzujú možnosti zavedených implemetácií [12].

- Neexistuje žiadny spôsob, ktorý by umožnil obijsť implicitnú alias analýzu implemetovanú v rozhraní. Vďaka možnému reťazeniu odpovedí však toto nepredstavuje výraznú komplikáciu.
- Jednotlivé priechody LLVM nemôžu informovať PassManager o zachovaní informácií relevatných pre alias analýzu. Bez tejto informácie však nie je možné vytvárať zložitejšie analýzy pracujúce nad celým modulom, ktoré potrebujú vystavávať points-to množiny pre zodpovedanie otázok.
- Transformačné priechody nemajú prostriedok akým by mohli alias analýzu informovať o zmenení zdrojového kódu. V konečnom dôsledku to len rozširuje problém uvedený v predchádzajúcom bode.

Základ algoritmu bol napísaný pre LLVM vo verzii 2.2. Odvtedy sa vývoj systému posunul ďalej a bolo nutné vo veľkej miere prepísať časti určené na integráciu algoritmu do systému LLVM a mechanizmy slúžiace na vystavanie obmedzení.

Hlavným problémom pre akúkoľvek analýzu, ktorá je postavená na vytvorení points-to množiny, je nemožnosť invalidácie už vytvoreného stavu automaticky, pomocou LLVM manažéra priechodov.

#### Prehľad

Algoritmus sa dá rozdeliť na dva velké celky. V inicializačnej fáze je vytvorená reprezentácia programu a sú ustanovené vzťahy medzi jednotlivými objektmi v programe.

- Prvým krokom je vytvorenie reprezentácie programu. Reprezentáciu vytvára metóda *IdentifyObjects*, ktorá vytvára *Node* objekt pre všetky globálne premenné a definície funkcií. Následne iteruje nad kódom každej funkcie a vytvára *Node* objekt pre každú alokáciu na hromade alebo zásobníku a zároveň pre premenné definované ako operandy.
- Po identifikovaní objektov sa vytvorí množina obmedzení pre Andersenov algoritmus. Iteruje sa nad všetkými inštrukciami programu a do obmedzenia pre každú inštrukciu, ktorá obmedzenie indukuje, je pridané do zoznamu. Implementované pomocou metódy *CollectConstraints*. Rieši sa aj pridávanie obmedzení pre inštrukcie, ktorých definícia nie je dostupná v danom zdrojovom kóde. Ich efekt sa modeluje pomocou *UniversalSet* obmedzenia. Zabezpečuje, že externe definovaná funkcia je braná ako nebezpečná pre vyhodnotenie noalias odpovede. Detailnejšie je implementácia obmedzení je popísaná v nasledujúcej časti.
- Vytvorenie pt množín pre jednotlivé objekty implementuje metóda *SolveConstraints*. Implementácia je založená na HCD, ktoré je popísané v časti 4.3.1. Je využitá heuristika zjednodušenia rozponávania cyklov. V článku [6] je popísaná kontrola cyklov pre každý uzol nezávisle, čo je náročné pri veľkých programoch. Heuristika vytvára zoznam uzlov, nad ktorým prebieha kontrola naraz.

Ak je reprezentácia postavená, alias analýza môže začať dodávať odpovede. Komunikácia so zvyškom systému prebieha pomocou sady metód definovaných v *AliasAnalysis* rozhraní.

- Metóda *alias* zisťuje, aký je medzi dvoma dodanými ukazateľmi vzťah. Ak ich pt množiny nemajú prienik, vracia odpoveď *noalias*. Ak aj prienik existuje, nasleduje skúmanie, či dané ukazatele neukazujú na rozličnú časť dátovej štruktúry.
- *pointsToConstantMemory* vracia *true* ak daný ukazateľ ukazuje na konštantnú globálnu premennú, funkciu alebo má hodnotu *null*.
- *getMustAliases* môže určiť, či daný ukazateľ musí ukazovať na danú lokáciu.
- *getModRefInfo* slúži na kontrolu, či dané volanie môže modifikovať alebo pristupovať k danej pamäťovej lokácii.

### Riešenie invalidácie

Invalidovanie vystvaných pt množín je možné dvoma spôsobmi. Ak priechod LLVM vyžaduje alias analýzu, tak po tom, ako od manažéru priechodov získa alias analýzu, zavolá inicializačnú funkciu. Jej parametrom je modul, nad ktorým priechod pracuje. Následne sa porovná dodaný modul s reprezentáciou modulu, ktorá bola použitá pri predchádzajúcom vystavaní pt množín a ak sa nezhodujú, tak prebehne vystavanie pt množín nanovo.

Je možné, že priechod vykoná zmenu, ktorá spôsobí zavedenie premennej, ktorá nemá reprezentáciu v pt množine. Preto sú tieto zmeny kontrolované aj pri každom vyžadovaní informácií od alias analýzy.

Prvé uvedené riešenie však vytvára veľkú výpočetnú náročnosť, nakoľko by bolo nutné vytvoriť novú reprezentáciu modulu pri akejkoľvek zmene. Keďže implementácia ignoruje manažér priechodov z hľadiska udržiavania vzťahov a závislostí, tak si udržiava vlastnú uplne separátnu inštanciu grafu volaní v module. Implicitne sa využíva druhá možnosť invalidácie, to znamená, že nová reprezentácia modulu je vystvaná, len ak dotaz na alias analýzu obsahuje lokáciu, ktorej reprezentácia nie je dostupná. Toto riešenie výrazne redukuje výpočetnú náročnosť a zároveň zachováva validitu reprezentácie.

### Implementácia obmedzení

Prepracovaná bola práca algoritmu s transformovaním kódu v LLVM IR na obmedzenia. Implementácia teraz plne využíva *InstructionVisitor*. Riešenie je využívané ak je potrebné vykonať špecifickú akciu pre rozličný typ inštrukcií. V praxi stačí aby priechod dedil triedu *InstVisitor* a reimplementoval metódy požadovaných tried inštrukcií. Visitor sa spúšťa zavolaním metódy *visit*. Navštívením každej inštrukcie sa vytvárajú obmedzenia algoritmu, tak ako sú uvedené v [2].

Inštrukcie *load*, *store* sú jednoznačne prepísané na pravidlá $\forall a \in pt(y).pt(a) \supseteq pt(x)$ a $\forall a \in pt(x).pt(y) \supseteq pt(a)$. Komplikovanejšie inštrukcie sú redukované podľa [7].

- *Cast* inštrukcia sa prepisuje na $pt(y) \supseteq pt(x)$, berie sa ako kopírovacie obmedzenie.
- *Select* vytvára dve závislosti a to $pt(y) \supseteq pt(x)$ a $pt(z) \supseteq pt(x)$ ak priradenie je do premennej $x$.
- *Phi* je špecializovanou inštrukciou slúžiacou na priradenie hodnoty do premennej, pričom táto hodnota môže pochádzať z rozličných základných blokov. Pomocou nej

sa riadi tok kontroly programom. Počet obmedzení sa vytvára podľa počtu argumentov, ktorých počet je variabilný, podľa toho, z koľkých základných blokov môže hodnota pochádzať. Vytvárajú sa kopírovacie obmedzenia analogicky k riešeniu *select* inštrukcie.

- *Vaarg* je inštrukcia, ktorá je využívaná na predávanie variabilného množstva argumentov volaniu funkcie. Využíva sa pri implementácii C makra *va_arg*. Transformuje sa znova na variabilné množstvo kopírovacích obmedzení.
- *Return* inštrukcia je využívaná na prepojenie pt množiny návratového objektu s pt množinou objektu, kam sa ukladá výsledok.
- *Call* pridáva obmedzenia pre inštrukciu volania, slúži na predanie pt množín argumentov k objektom nachádzajúcim sa v tele funkcie. Kopírovanie obmedzení sa týka skutočných argumentov definovaných pomocou *CallSite* a deje sa nezávisle na dátových typoch. Pretypovanie teda neovplyvňuje validnosť pt množiny.

### Implementácia popísaných rozšírení

Podstatným vylepšením pôvodného algoritmu [14] je modelovanie dátových štruktúr a prístupov k nim. Modelovanie je zabezpečené úpravou spracovania inštrukcie *GetElementPtr*. Jedná sa o inštrukciu, ktorá v LLVM IR zabezpečuje prístup k elementu premennej agregovaného dátového typu. Inštrukcia nepristupuje k dátam, iba poskytuje adresu.

**Príklad 8** Príklad využitia *GetElementPtr*

```
struct RT {
  char A;
  int B[10][20];
  char C;
};

int *foo(struct RT *s) {
  return &s[1].B[5][13];
}

Následne v LLVM:

%struct.RT = type { i8, [10 x [20 x i32]], i8 }
define i32* @foo(%struct.RT* %s) nounwind uwtable readnone optsize ssp {
entry:
  %arrayidx = getelementptr inbounds %struct.RT* %s, i64 1, i32 1,
              i64 5, i64 13
  ret i32* %arrayidx
}
```

V príklade 4.4.3 je vidieť sémantika inštrukcie *GetElementPtr*. Prvý operand identifikuje agregovanú premennú a jej typ, nasledujú operandy ktoré identifikujú konkrétny element štruktúry. Unifikovaný prístup k agregovaným dátovým štruktúram otvára priestor pre dodanie nových informácií ku konštruovaným obmedzeniam.

Pri vytváraní obmedzení pre inštrukciu *GetElementPtr* sa okrem klasického kopírovacieho obmedzenia $pt(y) \supseteq pt(x)$ vytvorí aj informácia o ceste k danému elementu $x$. Táto cesta je pripojená k uzlu, ktorý reprezentuje element $x$.

Ak metóda *alias* prijme dve lokácie, ktoré majú prienik v svojich pt množinách, tak nasleduje kontrola, či k nim je dostupná informácia o ceste k elementu dátovej štruktúry. Zároveň musí byť isté, že obe ukazujú na element, ktorý patrí premennej zhodného dátového typu.

V prípade ak indexami sú konštantné hodnoty je porovnanie priamočiare, ak sa nezhodujú, nemôžu ukázať na ten istý element. Problém nastáva pri vyhodnocovaní prístupu uvedeného v nasledujúcom príklade.

**Príklad 9** Príklad modelovania dátovej štruktúry

```
void fce(int **x,int *y){
int i,j;
for (j = 1; j <= 10 - 1; j++)
    for (i = j; i <= 10 - 1; i++){
        x[j][i] = x[j - 1][i - 1] + x[j][i];
    }
}
v LLVM:
%idx1 = sub i32 j , 1
%idx2 = sub i32 i , 1
%arrayidx1 = getelementptr inbounds int** %x, i64 idx1, i32 idx2
%arrayidx2 = getelementptr inbounds int** %x, i64 j, i32 i,
```

Algoritmus prejde cestu k elementom vytvorenú pre premenné *%arrayidx1* a *%arrayidx2*. Aj napriek tomu, že nevie, aká je hodnota indexov $i$ a $j$, vie určiť, že nedochádza k aliasu. Dôvodom je, že indexy sa líšia o známu konštantnú hodnotu.

Ďalším rozšírením je modelovanie inštrukcií *PtrToInt* a *IntToPtr*. Vo fáze určovania obmedzení je modelovaný efekt týchto inštrukcií. Slúžia na pretypovanie ukazateľového typu na celé číslo a naopak. Ak tieto inštrukcie nemodelujeme, tak musí byť určené, že ukazateľ vzniknutý využitím *IntToPtr* môže teoreticky ukazovať na akýkoľvek objekt v programe.

V špecifickej množine prípadov môže byť vytvorené kopírovacie obmedzenie. Podmienkou je, že vieme identifikovať cestu od pôvodného *PtrToInt* po *IntToPtr* a vieme určiť, že ukazujú do známeho objektu. Toto rozšírenie je tiež vhodné pri modelovaní agregovaných dátových štruktúr.

# Kapitola 5

# Predikované inštrukcie

Podľa [4] je predikovanie skupina architekturálnych a prekladových techník, ktorá sa zaoberá konvertovaním riadiacich závislostí na dátové závislosti. Dáva prekladaču možnosť obchádzať inštrukcie vetvenia. Predikácia vždy zahŕňa aj istú formu podmieneného vykonávania inštrukcií, založenú na boolovskom predikáte. Pod pojmom predikácia rozumieme všetky architekturálne techniky zaoberajúce sa odstránením inštrukcií vetvenia

## 5.1 Prístupy k predikácii inštrukcií

Na uvedenom príklade budeme ilustrovať dva možné prístupy k odstráneniu inštrukcie vetvenia z ukázaného kódu. Na ukážku slúži kód z VLIW architektúry VEX.

**Príklad 10** Príklad vetvenia v architektúre VEX.
Kód v jazyku C:

```
if (x > 0)
c = a * b; // 2 cyklová latencia
p[0] = c;
```

VEX kód:

```
(x=$r5, a=$r1, b=$r2, c=$r3, p=$r10)
cmpgt $b1 = $r5, 0
;;
br $b1, L1
;;
mpy $r3 = $r1, $r2;;
xnop 1 ## (predpokladáme 2 cyklovú latenciu)
;;
L1:
stw 0[$r10] = $r3
;;
```

### 5.1.1 Plná predikácia

Plná predikácia znamená, že sa podmienečne vykonajú všetky operácie v obidvoch vetvách podľa podmienky odvodenej z inštrukcie vetvenia. Vyžaduje dodanie extra boolovského predikátu ku všetkým inštrukciám z inštrukčnej sady. Nutné je aj vytvorenie sady inštrukcií definujúcich predikáty a banku 1-bitových registrov predikátov. V príklade je uvedená predikovaná inštrukcia násobenia.

**Príklad 11** Rozšírenie VEXu o plnú predikáciu.

```
## Notácia ,,($p) op‘‘ znamená ,, if(p) op‘‘
##
cmpgt $p1 = $r5, 0
;;
($p1) mpy $r3 = $r1, $r2
xnop 1 ## (predpokladáme 2 cyklovú latenciu)
;;
stw 0[$r10] = $r3
;;
## ak p1 je TRUE, tak sa vykoná mpy
## ak je p1 FALSE, tak sa mpy považuje za nop
```

### 5.1.2 Čiastočná predikácia

Vykonajú sa všetky inštrukcie v obidvoch vetvách programu. Po vykonaní oboch vetiev sa vyberie finálna hodnota ovplyvnených premenných. Výber je založený na podmienke odvodenej z inštrukcie vetvenia. Čiastočná predikácia vyžaduje niekoľko nových inštrukcií ( najmä podmienený *move* alebo *select*). Tento prístup má viac obmedzení avšak nevyžaduje rozšírenie formátu operandov.

**Príklad 12** Rozšírenie VEXu o čiastočnú predikáciu. Príklad ilustruje využitie inštrukcie select a podmienený move.

```
## Využívame VEX slct inštrukciu.
##
mpy $r4 = $r1, $r2
;;
cmpgt $b1 = $r5, 0
;;
slct $r3 = $b1, $r4, $r3
;;
stw 0[$r10] = $r3
;;
## Inštrukcia mpy je špekulatívne vykonaná vždy
## ak b1 je TRUE, $r3 má novú hodnotu ($r4)
## ak b1 je FALSE, $r3 si ponechá starú hodnotu
## Nasleduje ekvivalentný kód s˜podmienou inštrukciou move
##
mpy $r4 = $r1, $r2
;;
cmpgt $b1 = $r5, 0
;;
cmov $r3 = $b1, $r4
;;
stw 0[$r10] = $r3
;;
```

### 5.1.3 Prínos predikácie

Predikácia umožňuje prekladaču lepšie využiť ILP architektúry. Paralelizmus je však umelo vytvorený. Iba jedna vetva vytvára užitočné výsledky, avšak vykonávajú sa inštrukcie vo všetkých vetvách.

Značne urýchluje programy so zložitými vetveniami, eliminuje zložitú kompenzáciu kódu. Urýchlenie je významné najmä pri architektúrach s hlbokou pipelinou. Významné vylepšenie predstavuje predikcia aj pre dátovo intenzívne programy, určené na triedenie, kompresiu, prácu s databázami.

Z hľadiska prekladača umožňuje predikácia pracovanie s väčšími základnými blokmi programu. Prínos má pre optimalizácie cyklov a je to prístup, ktorý podporuje software-pipelining pri cykloch so zložitým riadením.

Najefektívnejšia je pre programy, pri ktorých je počet vykonaní jednotlivých vetiev vyvážený. Plná predikácia však zvyšuje zložitosť architektúry, umožňuje predikáciu blokov s akoukoľvek kombináciou inštrukcií. Čiastočná predikácia je menej náročná na implementovanie, avšak v kombinácii s agresívnou špekuláciou, môže mať podobné výsledky ako plná predikácia [4].

## 5.2 If-konverzia

V rámci svojej diplomovej práce implementujem postup známy ako if-konverzia. Ide o jednu z hlavných techník vyvinutých na podporu predikcie. Transformuje riadiace závislosti na dátové závislosti a konvertuje acyklickú podmnožinu CFG (graf toku riadenia) z nepredikovanej reprezentácie na postupnosť predikovaných operácií.

Pre každý blok sa vytvorí predikátová premenná, ktorá je pravdivá iba ak je blok operácií vykonaný. Následne sa prepíše každá operácia na predikovaný ekvivalent. Prakticky ide o transformovanie podmienok na definície predikátových premenných. Netriviálnosť spočíva v zlučovaní ciest grafom toku riadenia.

Pod if-konverziou môžeme rozumieť aj postup zlučuje acyklické podgrafy grafu toku kontroly, ktoré vytvára konštrukcia *if* v LLVM IR. Zlúčenie prebieha tak, aby boli vypočítané obidve vetvy a rozhodovanie pomocou podgrafu je nahradené inštrukciou *select*. Základná charakteristika ostáva splnená, riadiaca závisloť je konvertovaná na dátovú závislosť.

### 5.2.1 LLVM a if-konverzia

LLVM framework podporuje if-konverziu nad *Machine code*, čo je reprezentácia programu v backende, využívaná pri generovaní kódu. Implementované riešenie plne zodpovedá postupu popísanému v 5.2 Podpora if-konverzie na úrovni LLVM IR však neexistuje. V rámci projektu LLVM prebiehala diskusia o implementovaní if-konverzie aj na úrovni LLVM IR, avšak implementácia nebola realizovaná. Nasledujúca časť je venovaná návrhu a implementácii profilom riadenej if-konverzie na úrovni LLVM IR.

### 5.2.2 Profilom riadené optimalizácie

Napriek tomu, že v LLVM je implementovaná if-konverzia, existuje stále priestor pre zlepšenie výkonnosti. Implementovaný algoritmus nemá žiadnu informáciu o reálnom behu programu.

Zlepšenie výkonu môže nastať ak je optimalizácia schopná získať informácie o tom, ako často sa konkrétna vetva v podgrafe, ktorý má byť transformovaný vykonáva. Ak vďaka profilu vieme určiť, že istá vetva sa v čase behu vykonáva oveľa častejšie ako iná, tak vieme zabrániť tomu, aby nastala nevýhodná transformácia kontrolnej závislosti na dátovú na úrovni LLVM IR. Ak by prebehla transformácia zlúčením rozličných vetiev, tak by bolo nutné vyhodnocovať inštrukcie, ktoré by v opačnom prípade neboli vykonané.

Vďaka profilovacím informáciám získaným z behu programu vieme získať dostatok informácií na to, aby sme zabránili zbytočnej transformácii, ktorá by spôsobila nadbytočné vykonávanie kódu.

#### Typy profilu

Podľa [5] vieme určiť tri základné typy profilov.

- Profily toku riadenia programu sa zaoberajú sledovaním cesty vykonávania programu grafom toku kontroly. Cesta je reprezentovaná poradím uzlov, ktoré predstavujú navštívené základné bloky.

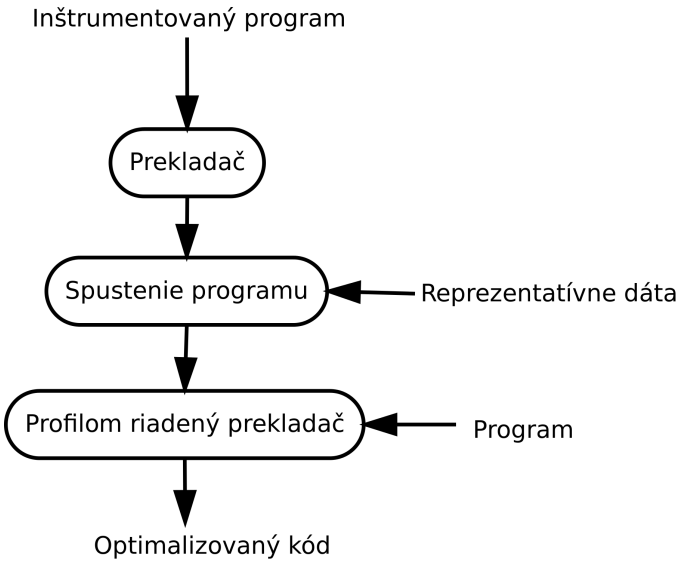
  Vďaka tomuto typu profilu vieme vypočítať ako často bol vykonaný podgraf grafu toku kontroly. Profily toku riadenia sa ďalej delia na niekoľko základných druhov.

  - Profily uzlov poskytujú frekvencie vykonania základného bloku v grafe toku kontroly.
  - Profily hrán však na rozdiel od predchádzajúcich poskytujú frekvencie vykonania hrán v grafe toku kontroly. Obidva typy sú porovnateľné, avšak z profilu hrán sa vždy dá vypočítať profil uzlov. Opačným smerom to však nie je vždy možné.

- Profily hodnôt identifikujú špecifické hodnoty, zaznamenané ako operandy inštrukcií a zároveň ich frekvencie výskytu. Vďaka tejto informácii môže prekladač rozpoznať hodnoty, ktoré sú takmer konštantné a využiť túto informáciu na vykonanie špecifických optimalizácií. Napríklad redukciu sily, premiestnenie skoro invariantného kódu mimo cyklus alebo zjednodušovanie konštánt.

- Profily adries môžu byť získané vo forme množín adries v pamäti, ktoré sú referencované programom. Väčšinou sú využité na na optimalizovanie rozloženia dát, alebo pre zvyšovanie výkonu hierarchie pamätí. Používajú sa aj na zlepšovanie prístupu ku globálnym dátam a dátam uloženým na hromade.

### Preklad pomocou profilom riadenej optimalizácie




Obrázek 5.1: Znázornenie procesu profilom riadeného prekladu. Obrázok je inšpirovaný [5].

Proces profilom riadeného prekladu je znázornený na obrázku 5.1. Predtým ako môže byť spustená profilom riadená optimalizácia, inštrumentovaný program musí byť spustený nad sadou reprezentatívnych vstupov. Vygenerované profilovacie informácie sú využité pri novom preklade, ktorý generuje optimalizovaný kód.

### Využitie profilu v LLVM

Systém LLVM poskytuje podporu pre využitie profilácie. Programom *opt* je vložená do programu v LLVM IR inštrumentácia. Prakticky sa vložia inštrukcie, ktoré inkrementujú čítač

pri každom priechode cez základný blok. Vďaka tomu je následne možné získať informáciu o počte priechodov cez uzly grafu toku riadenia.

V systéme LLVM sú podporované štyri základné typy profilov.

- Profil hrán je hranový profil zodpovedajúci definícii v cite.
- GCOV profil zodpovedá profilu využívanému v nástroji *gcov*, ktorý je súčasťou *GCC*.
- Optimalizovaný profil hrán.
- Profil cesty využíva inštrumentáciu vhodnú pre Ball-Larusove profilovanie. Je založené na hladaní najfrekventovanejších acyklických ciest grafom toku riadenia.

Po vložení inštrumentácie do zvoleného programu je nutné daný program spustiť s vhodnými vstupnými dátami. Pre čo najpouižitelnejšie profilovacie informácie, je vhodné aby vstupné dáta pre program čo najviac reprezentovali dáta nad ktorými bude pracovať preložený program. Program sa dá interpretovať nástrojom *lli*, ktorý patrí do systému LLVM. Tak isto sa dá vygenerovať strojový kód a program simulovať pomocou simulátoru dostupného v Codasip frameworku. Profil má podobu súboru *llvmprof.out*. Nástroje v rámci systému LLVM s profilmi pracujú pomocou triedy *ProfileInfoLoader*.

### 5.2.3 Návrh a implementácia

Algoritmus if-konverzie a spôsob profilom riadeného vytvárania zoznamu vhodných transformácií je integrovaný v nástroji *opt-superblock*. Zastrešuje sadu algoritmov vhodných pre kompiláciu pre VLIW architektúry. Jedná sa najmä o konštrukciu a plánovanie superblokov, podporu bundlovania inštrukcií. Implementácia je rozdelená do dvoch hlavných celkov.

#### Identifikácia vhodných Phi inštrukcií

V prvom kroku je nutné identifikovať vhodné podgrafy v grafe toku riadenia programu. Keďže phi inštrukcia predstavuje spôsob akým LLVM IR modeluje riadiace štruktúry, je nutné sa pri analýze zamerať práve na bloky, z ktorých sa kontrola predáva *phi* inštrukcii.

Implementáciu zabezpečuje trieda *PreparePTSProfile*. Ide o LLVM *ModulePass* priechod. Pracuje nad celým modulom, pričom iteruje cez definície všetkých funkcií. Postupne navštívi všetky základné bloky a získava informáciu o frekvencii vykonania hrán grafu toku riadenia.

Informácie sú získavané z vytvoreného profilu pomocou triedy *ProfileInfoLoader*. Zároveň prebieha kontrola konzistencie medzi informáciami z profilu a skutočným kódom.

Ak narazí na *phi* inštrukciu, tak sa následne vyhodnotia dve vlastnosti.

- Počet vykonaní hrán vstupujúcich do *phi* inštrukcie.
- Veľkosť kódu základného bloku, z ktorého prechádza kontrola do *phi* inštrukcie.

Či je závislosť vhodná na transformáciu určuje nie len počet vykonaní hrany. Testovaním na sade benchmarkov bolo zistené, že rozdiel medzi počtom vykonaní by nemal presiahnuť 30%. Zároveň však transformované bloky nemôžu byť príliš veľké, nakoľko odstránenie závislosti by neprinieslo zväčšenie, ktoré by prekonalo stratu spôsobenú vykonávaním inštrukcií vo všetkých vstupných blokoch.

Výsledkom priechodu je množina *phi* inštrukcií, ktoré sú vhodné k prípadnej transformácii na dátovú závislosť. Inštrukcia je identifikovaná menom funkcie v kotrej sa nachádza

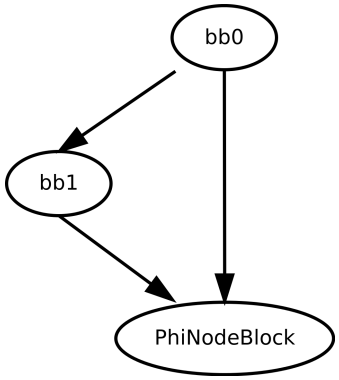
a identifikátorom základného bloku, ktorý uvádza. O tom či sa konkrétna závislosť dá transformovať rozhoduje transformačný priechod, splnené musia byť ďalšie podmienky.

### Transformácia grafu toku kontroly

Samotná transformácia na dátové závislosti je implementovaná ako priechod

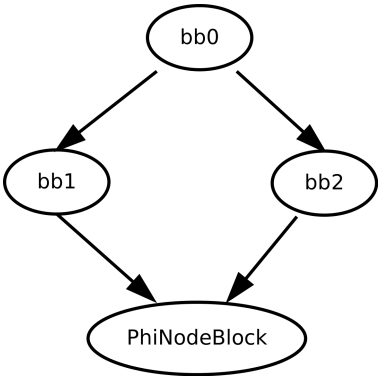
*TransformPhiInstrToSelect*, ktorý pracuje nad celým modulom. Iteruje nad definíciou každej funkcie v programe. Ak identifikuje *phi* inštrukciu, ktorá uvádza základný blok, tak prebieha pokuso transformáciu.

- Identifikovaný základný blok musí byť zhodný s jedným z blokov, ktoré identifikovala *PreparePTSProfile*.
- Nasleduje kontrola, zisťujúca, či je daný podgraf vytvorený *phi* inštrukciou vhodný k transformácii. Existujú dva vhodné vzory.
  - Trojuholníkový vzor.




Obrázek 5.2: Pri identifikovaní tohto vzoru je podstatné aby mal základný bok *bb0* práve dvoch nasledovníkov, zatiaľ čo základný blok *bb1* má iba jedneho.

  - Kosoštvorcový vzor.




Obrázek 5.3: Alternatíva k trojuholníkovému vzoru, pri ktorej sa tiež dajú zjednotiť základné bloky a hodnoty prichádzajúce z *bb1* a *bb2* priradiť pomocou inštrukcie *select*.

- Posledným krokom je samotná transformácia. Ak *phi* inštrukcia má len práve dva vstupné body, ktoré sú identifikované vzorom, nasleduje zlúčenie základných blokov daného vzoru. Na to slúži *MergeBlockIntoPredecessor*. Z pôvodnej *phi* inštrukcie sú vstupné premenné izolované a vložené ako operandy novej inštrukcie *select*. Uvedené vhodné podgrafy sú teda zlúčené do jednoho základného bloku.

Implementovaný algoritmus je využitý v nástroji *opt-superblock*. Na správne fungovanie je nutné nástroju dodať profilačné informácie. If-konverzia je zaradená tak, aby bola vykonávaná pred spúšťaním transformácií, ktoré ovplyvňujú graf toku kontroly a mohli by teda narušiť vzory, ktoré sa menia na dátové závislosti.

# Kapitola 6

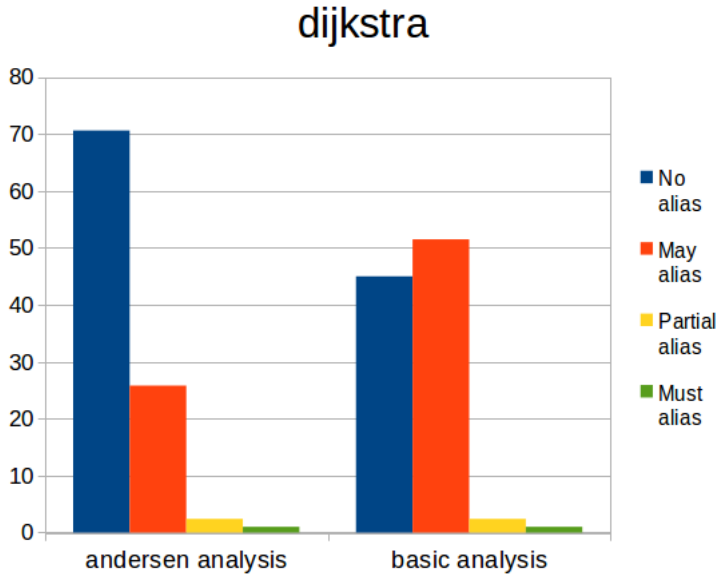
# Dosiahnuté výsledky

## 6.1 Alias analýza

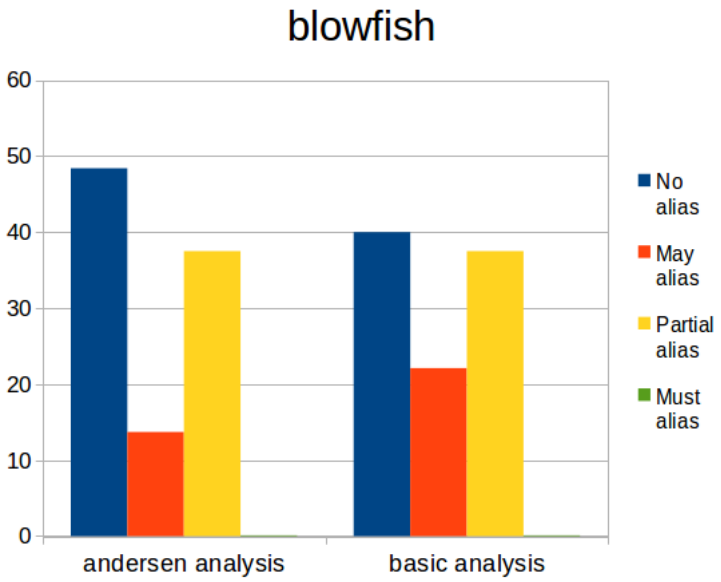
Funkčnosť alias analýzy bola testovaná nad testami pochádzajúcimi zo známych benchmarkov. Mimo iných *full-retval-gcctestsuite* a *LLVM testsuite*. Testy alias analýzy sú rozdelené do dvoch častí. V prvej je znázornené o koľko sa zvýšila presnosť odpovedí alias analýzy oproti už existujúcim algoritmom. Druhá časť sa venuje efektom novej alias analýzy na preložený program.

### 6.1.1 Presnosť

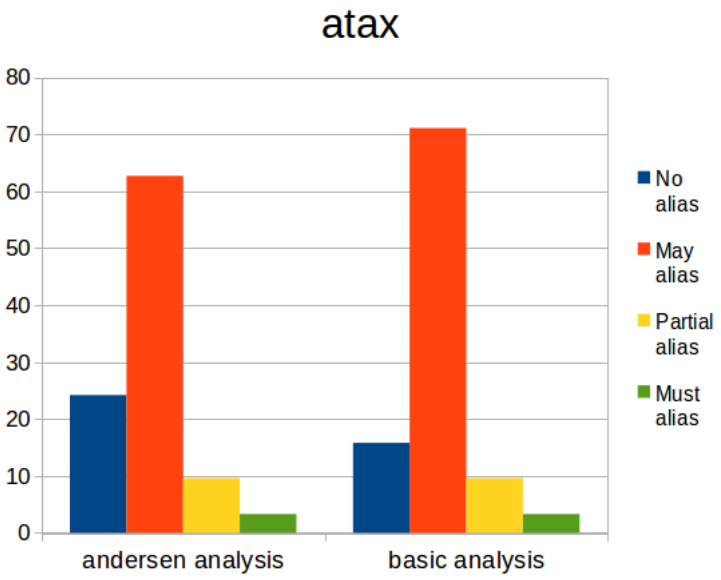
Na meranie presnosti odpovedí alias analýzy sa v systéme LLVM využíva nástroj *opt* spustený s parametrom *-aa-eval*. Nástroj sa spýta alias analýzy na dostupné informácie o všetkých dvojiciach ukazateľov, ktoré sú v programe. Následne vystavia správu o percentuálnom zastúpení jednotlivých odpovedí.



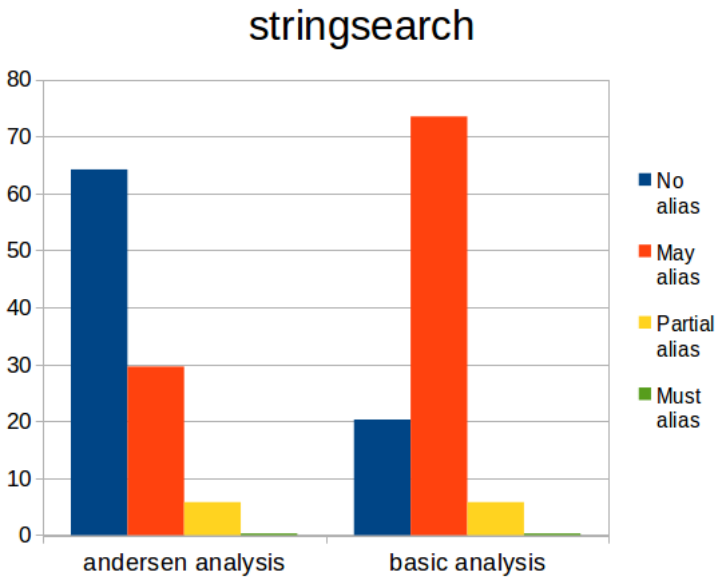

Obrázek 6.1: Percentuálna miera zastúpenia jednotlivých odpovedí pri použití základnej alias analýzy v LLVM a pri použití implementovaného algoritmu.




Obrázek 6.2: Percentuálna miera zastúpenia jednotlivých odpovedí pri použití základnej alias analýzy v LLVM a pri použití implementovaného algoritmu.




Obrázek 6.3: Percentuálna miera zastúpenia jednotlivých odpovedí pri použití základnej alias analýzy v LLVM a pri použití implementovaného algoritmu.




Obrázek 6.4: Percentuálna miera zastúpenia jednotlivých odpovedí pri použití základnej alias analýzy v LLVM a pri použití implementovaného algoritmu.

Z uvedených príkladov je zrejmé, že implementovaný algoritmus alias analýzy je výrazne presnejší ako algoritmus štandardne používaný nástrojmi LLVM. Väčší podiel odpovedí *no alias*, je výhodnejší pre transformácie využívajúce alias analýzu. Zložitejšie algoritmy a algoritmy intenzívnejšie využívajúce ukazatele vykazujú najväčšie zvýšenie presnosti odpovedí.

Ďalší vývoj možno zamerať na zlepšenie modelovania dátových štruktúr a vylepšenie prepisovania vybraných inštrukcií na obmedzenia. Zlepšenie modelovania dátových štruktúr je dosiahnuteľné identifikáciou komplikovanjších prístupov k ich elementom.

### 6.1.2 Urýchlenie výsledného kódu

Na vybraných programoch z použitých testovacích sád je ukázaný efekt využitia presnejšej alias analýzy na výsledný kód. Meranenie je vykonané pomocou simulátoru procesoru CodixVLIW. Jedná sa o VLIW procesor, dodávaný firmou Codasip, schopný vykonávať štyri inštrukcie zároveň. Uvedené programy pochádzajú z rozličných sád benchmarkov dostupných v LLVM testsuite. Reprezentované sú programy zo sád *Polybench*, *Mediabench* a *MiBench-network*.

Tabulka 6.1: Trvanie programu je uvedené ako počet cyklov simulátoru procesoru nutných na vykonanie programu.

| Program | Pôvodný kód | Optimalizovaný kód | Percentuálne urýchlenie |
|---|---|---|---|
| blowfish | 8489546 cyklov | 8390186 cyklov | 1.2% |
| stringsearch | 7083796 cyklov | 6750771 cyklov | 5% |
| mpeg2 | 28570367 cyklov | 25876596 cyklov | 9.5% |
| atax | 198898 cyklov | 196657 cyklov | 2% |
| dynprog | 710673 cyklov | 679453 cyklov | 5% |
| 3mm | 7044191 cyklov | 7035959 cyklov | 0.2% |
| dijkstra | 264768784 cyklov | 257032486 cyklov | 3% |

Efekt vylepšenej alias analýzy je viditeľný najmä na rozsiahlejších programoch, kde transformácie v systéme LLVM vedia viac ťažiť z presnejších informácií poskytovaných alias analýzou. Pri menších programoch sa tento efekt stráca, preto ďalšia činnosť je smerovaná na vylepšenie práce transformácií s informáciami od alias analýzy.

## 6.2 If-konverzia

Funkčnosť implementovaného algoritmu profilom riadenej if-konverzie bola overená testovaním nad sadou benchmarkov. Uvedené merania pochádzajú zo simulovania vybraných testov na simulátore procesoru CodixVLIW.

Tabulka 6.2: Trvanie programu je uvedené ako počet cyklov simulátoru procesoru nutných na vykonanie programu.

| Program | Pôvodný kód | Optimalizovaný kód | Percentuálne urýchlenie |
|---|---|---|---|
| crc.c | 2700060 cyklov | 2409435 cyklov | 11% |
| sha.c | 2171 cyklov | 1952 cyklov | 10% |
| bitcount.c | 35634 cyklov | 36400 cyklov | -2% |
| isqrt.c | 285 cyklov | 228 cyklov | 20% |
| dijkstra.c | 834793 cyklov | 523795 cyklov | 38% |

Zlepšenie výkonu pri použití optimalizácie nastalo vo väčšine uvedených prípadov. Avšak existujú programy, v ktorých transformácia na dátové závislosti zhorší výsledný kód. Rezervy algoritmu ostávajú najmä vo vyhodnocovaní ceny vykonávania kódu v zlúčených základných blokoch. Vďaka využitiu profilovacích informácií však nastáva malé množstvo prípadov, v ktorých je odhad nesprávny.

# Kapitola 7

# Záver

Výsledkom diplomovej práce je rozšírený algoritmus alias analýzy a implementovaná profilom riadená if-konverzia. Andersenov algoritmus bol rozšírený o modelovanie dátových štruktúr a pokročilú transformáciu obmedzení LLVM IR špecifických inštrukcií. Vďaka týmto rozšíreniam algoritmus s *flow-insensitive* vlastnosťami získava prehľad o dátových štruktúrach využitých v programe a zvyšuje sa presnosť informácií dostupných o ukazateľoch.

Implementovaním if-konverzie na úrovni LLVM IR vzniká možnosť transformácie kontrolných závislostí na dátové a vďaka využitu profilácie je značne obmedzené množstvo transformácií, ktoré by priniesli spomalenie.

Riešenia boli otestované nad sadou benchmarkov a bolo preukázané urýchlenie testovaných programov. Implementovaný a vylepšený algoritmus alias analýzy je výrazne presnejší ako štandardné riešenie v LLVM. Integrácia alias analýzy pracujúcej nad celým modulom bol netriviálny problém, ktorý vyžadoval modifikáciu zvoleného algoritmu, ale aj zmeny na strane LLVM. Analýza je plne integrovaná a použiteľná vo všetkých nástrojoch LLVM. Ostáva však priestor na zlepšenie implementovaných riešení a vylepšenie využívania informácií poskytovaných alias analýzou.

Jadro činnosti spočívalo v hľadaní optimálneho algoritmu alias analýzy pre LLVM, práci na jeho vylepšení a implementovaní if-konverzie, ktorá by nespôsobovala reálne spomalenie preloženého kódu. Riešenia sú plne nasadené a používané vo frameworku Codasip. Cennou skúsenosťou bol návrh a implementácia netriviálnych algoritmov tak, aby boli prakticky využiteľné.

# Literatura


[1] Webové stránky frameworku Codasip. `http://www.codasip.com/`.

[2] Andersen, L.; Institut, K. U. D.: *Program Analysis and Specialization for the C Programming Language: Ph.D. Thesis*. DIKU rapport, Datalogisk Institut, Københavns Universitet, 1994.

[3] Brown, A.; Wilson, G.: *The Architecture of Open Source Applications*. 2011, ISBN 978-1-257-63801-7, 432 s.

[4] Fisher, J. A.; Faraboschi, P.; Young, C.: *Embedded computing - a VLIW approach to architecture, compilers, and tools.* Morgan Kaufmann, 2005, 671 s., iSBN 978-1-55860-766-8.

[5] Gupta, R.; Mehofer, E.; Zhang, Y.: Profile Guided Compiler Optimizations. 2002.

[6] Hardekopf, B.; Lin, C.: The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. *SIGPLAN Not.*, ročník 42, č. 6, Červen 2007: s. 290–299, ISSN 0362-1340, doi:10.1145/1273442.1250767.

[7] Hind, M.: Pointer Analysis: Haven'T We Solved This Problem Yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, New York, NY, USA: ACM, 2001, ISBN 1-58113-413-4, s. 54–61, doi:10.1145/379605.379665.

[8] Lattner, C.: The LLVM Target-Independent Code Generator [online]. `http://llvm.org/docs/CodeGenerator.html`, 2012-04-19 [cit. 2014-01-05].

[9] Lattner, C.: LLVM Alias Analysis Infrastructure [online]. `http://llvm.org/docs/AliasAnalysis.html`, 2012-04-19 [cit. 2014-01-08].

[10] Lattner, C.; Adve, V.: The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign, 2002-08-02.

[11] Lattner, C.; Adve, V.: LLVM Language Reference Manual [online]. `http://llvm.org/docs/LangRef.html`, 2012-04-19 [cit. 2014-05-04].

[12] Lattner, C.; Laskey, J.: Writing an LLVM Pass [online]. `http://llvm.org/docs/WritingAnLLVMPass.html`, 2012-04-19 [cit. 2014-01-05].

[13] Lattner, C.; Lenharth, A.; Adve, V.: Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World. *SIGPLAN Not.*, ročník 42, č. 6, Červen 2007: s. 278–289, ISSN 0362-1340, doi:10.1145/1273442.1250766.

[14] Lhoták, O.; Chung, K.-C. A.: Points-to Analysis with Efficient Strong Updates. *SIGPLAN Not.*, ročník 46, č. 1, Leden 2011: s. 3–16, ISSN 0362-1340, doi:10.1145/1925844.1926389.

[15] Morgan, R.: *Building an Optimizing Compiler*. Butterworth-Heinemann, 1998, ISBN 9781555581794.

[16] Muchnick, S. S.: *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, iSBN 1-55860-320-4.

[17] Spencer, R.; Henriksen, G.: LLVM's Analysis and Transform Passes [online]. `http://llvm.org/docs/Passes.html`, 2012-04-19 [cit. 2014-01-10].

[18] Srikant, Y. N.; Shankar, P.: *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. Boca Raton, FL, USA: CRC Press, Inc., druhé vydání, 2007, ISBN 142004382X, 9781420043822.

[19] Steensgaard, B.: Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, New York, NY, USA: ACM, 1996, ISBN 0-89791-769-3, s. 32–41, doi:10.1145/237721.237727.